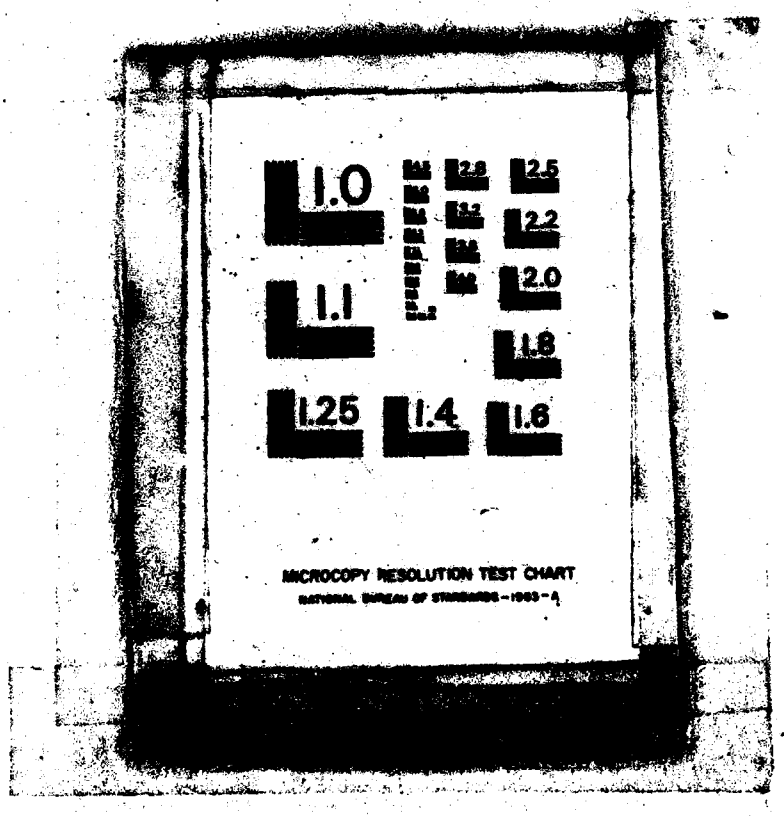
AD-A135 201 THE DISTRIBUTED V KERNEL AND ITS PERFORMANCE FOR DISKLESS WORKSTATIONS(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE D R CHERITON ET AL. JUL 83 1/1

UNCLASSIFIED STAN-CS-83-973 MDA903-80-C-0102 F/G 17/2 NL


1.0
1.1
1.25
1.4
1.6
2.0
2.2
2.5
MICROCOPY RESOLUTION TEST CHART

July 1983

Report No. STAN-CS-83-973
Also numbered CSL 246

The Distributed V Kernel and Its Performance for Diskless Workstations

by

David R. Cheriton and Willy Zwaenepoel

Contract [illegible]

Department of Computer Science

Stanford University
Stanford, CA 94305

The Distributed V Kernel and its Performance for Diskless Workstations

David R. Cheriton and Willy Zwaenepoel

Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University

The author's

Abstract

The distributed V kernel is a message-oriented kernel that provides uniform local and network interprocess communication. It is primarily being used in an environment of diskless workstations connected by a high-speed local network to a set of file servers. This paper is a performance evaluation of the kernel, with particular emphasis on the cost of network file access. Our measurements show that over a local network:

1. Diskless workstations can access remote files with minimal performance penalty.
2. The V message facility can be used to access remote files at comparable cost to a well-tuned special-purpose file access protocol.

It is concluded that it is feasible to build a distributed system with all network communication using the V message facility even when most of the network nodes have secondary storage.

1. Introduction

Ethernet [7]. Network interprocess communication is predominantly used for remote file access since most SUN workstations at Stanford are configured without a local disk.

This paper reports our experience with the implementation and use of the V kernel. Of particular interest are the consequential aspects of our approach, namely:

1. The use of diskless workstations with all secondary storage provided by backend file servers.
2. The use of a general purpose network interprocess communication facility for access to special-purpose file access protocols and, in particular, the use of a Thoth-like interprocess communication mechanism.

The more conventional approach is to configure workstations with a small local disk, using network-based file servers for paging, program, system temporary, home, [illegible] of significant number.

1. Higher hardware cost per workstation.
2. Higher maintenance and operation of units with shared file servers.
3. Little or no memory or processing overhead on the [illegible] file system and disk handling.
4. Fewer problems with installation, maintenance and operation of disk.

The major disadvantage is the overhead of providing all file

well as relatively easy to use, does not support application-level use of streaming.

These potential problems prompted a performance evaluation of our methods, with particular emphasis on the efficiency of file access. This emphasis on file access distinguishes our study from similar studies [10, 13]. The results of our study strongly support the idea of building a distributed system using diskless workstations connected by a high-speed local network to one or more file servers. Furthermore, we show that remote file access using the V kernel IPC facility is only slightly more expensive than a lower bound imposed by the basic cost of network communication. From this we conclude that relatively little improvement in performance can be achieved using protocols further specialized to file access.

2. V Kernel Interprocess Communication

The basic model provided by the V kernel is that of many small processes communicating by messages. A process is identified by a 32-bit globally unique *process identifier* or *pid*. Communication between processes is provided in the form of short fixed-length messages, each with an associated reply message, plus a data transfer operation for moving larger amounts of data between processes. In particular, all messages are a fixed 32 bytes in length.

The common communication scenario is as follows. A client process executes a *Send* to a server process which then completes execution of a *Receive* to obtain the message and eventually executes a *Reply* to respond with a reply message that is sent to the client. We refer to this sequence as a *message exchange*. The receiver may examine any of some fixed or specified data transfer operations between the time the message is received and the time the reply message is sent.

The following section describes the primitives relevant to this paper. The [illegible] section is adapted from [7]. The kernel manual [8] is a complete specification of the kernel facilities.

2.1. Primitives

Send(message, pid)

Send the message to the process identified by pid, blocking the sender until the receiver executes a Receive and then a Reply or ReplyWithSegment specifying the sender's pid. On return, the message buffer is overwritten with the reply message.

ReceiveWithSegment(message, segptr, segsize)

Block until a message is received, then return with the message. If a segment is specified in the message with read access, up to the first segsize bytes of the segment are also returned in the array starting at segptr, with *segptr* specifying the actual number of bytes received.

Reply(message, pid)

Send a 32-byte reply contained in the message buffer to the specified process providing it is awaiting a reply from the replier. The sending process is readied upon receiving the reply; the replying process does not block.

ReplyWithSegment(message, pid, destptr, segptr, segsize)

Send a reply message as done by *Reply* but also transmit the data segment specified by segptr and segsize to *destptr* in the destination process' address space.

MoveFrom(srcpid, dest, src, count)

Copy *count* bytes from the segment starting at *src* in the address space of *srcpid* to the segment starting at *dest* in the calling process's space. The *srcpid* must be awaiting reply from the calling process and must have provided read access to the segment of memory in its address space using the message conventions described under *Send*.

MoveTo(destpid, dest, src, count)

Copy *count* bytes from the segment starting at *src* in the active process's space to the segment starting at *dest* in the address space of the *destpid* process. The *destpid* must be awaiting reply from the active process and must have provided write access to the segment of memory in its address space using the message conventions described under *Send*.

SetPid(logicalid, pid, scope)

Associate *pid* with the specified *logicalid* in the specified scope, which is one of local, remote or both. *Scope* is advisory only; [illegible] etc.

pid = GetPid(logicalid, scope)

Return the process identifier associated with *logicalid* in the specified scope if any, else 0.

[illegible]

must be allocated in the kernel and large amounts of data are transferred directly between users' address spaces without extra copies. Moreover, by virtue of the synchrony of the communication, the kernel's message buffers can be statically allocated. As exemplified in Thoth, these factors make for a small, efficient kernel.

The V message primitives appear ill-suited in several ways for a network environment, at least on first observation. The short, fixed-length messages appear to make inefficient use of large packet sizes typically available on local networks. The synchronous nature of the message sending would seem to interfere with the true parallelism possible between separate workstations. And the economies of message buffering afforded by these restrictions in a single machine implementation are less evident in a distributed environment. Finally, the separate data transfer operations *MoveTo* and *MoveFrom* appear only to increase the number of remote data transfer operations that must be implemented in the distributed case.

However, our experience has been that the V message primitives are easily and efficiently implemented over a local network. Moreover, we have found that the semantics of the primitives facilitated an efficient distributed implementation. The only major departure from Thoth was the explicit specification of segments in messages and the addition of the primitives *ReceiveWithSegment* and *ReplyWithSegment*. This extension was done for efficient page-level file access although it has proven useful under more general circumstances, e.g. in passing character string names to name servers.

3. Implementation Issues

A foremost concern in the implementation of the kernel has been efficiency. Before describing some of the implementation details of the individual primitives, we list several aspects of the implementation that are central to the efficient operation of the kernel.

1. Remote operations are implemented directly in the kernel instead of through a process-level network server. [illegible]

[illegible]

internet functionality and local net performance, we have chosen not to burden the dominant (local net) operation with any more overhead than is strictly necessary.

3. The synchronous request-response nature of a reply associated with each message is exploited to build reliable message transmission directly on an unreliable datagram service, i.e. without using an extra layer (and extra packets) to implement reliable transport. The reply message serves as an acknowledgement as well as carrying the reply message data.
4. The mapping from process id to process location is aided by encoding a host specification in the process identifier. The kernel can thus determine quickly whether a process is either local or remote, and in the latter case on which machine it resides.
5. There are no per-packet acknowledgements for large data transfers (as in *MoveTo* and in *MoveFrom*). There is only a single acknowledgement when the transfer is complete.
6. File page-level transfers require the minimal number of packets (i.e. two) because of the ability to append short segments to messages using *ReceiveWithSegment* and *ReplyWithSegment*.

The following sections look at particular aspects of the implementation in greater detail.

3.1. Process Naming

V uses a global (flat) naming space for specifying processes, in contrast to the local port naming used in DEMOS [1] and Accent [12]. Process identifiers are unique within the context of a local network. On the SUN workstation, it is natural for the V kernel to use 32-bit process identifiers. The high-order 16 bits of the process identifier serve as a *logical host identifier* subfield while the low-order 16 bits are used as a locally unique identifier.

In the current 3 Mb Ethernet implementation, the top 8 bits of the logical host identifier are the physical network address of the workstation, making the process identifier to network address mapping trivial. In the 10 Mb implementation, a table maps logical hosts to network addresses. When there is no table entry for the specified logical host, the message is broadcast. [illegible]

[illegible]

process. With the exception of *GetPid*, kernel operations with no process identifier parameters are implicitly local to the workstation.

GetPid uses network broadcast to determine the mapping of a logical process identifier to real process identifier if the mapping is not known to the local kernel. Any kernel knowing the mapping can respond to the broadcast request. The addition of local and remote scopes was required to discriminate between server processes that serve only a single workstation and those that serve the network.

3.2. Remote Message Implementation

When a process identifier is specified to *Send* with a logical host identifier different from that of the local machine, the local pid validation test fails and *Send* calls *NonLocalSend* which handles transmission of the message over the network.

The *NonLocalSend* routine writes a interkernel packet on the network addressed to the host machine of this process or else broadcasts the packet if the host machine is not known. When the host containing the recipient process receives the packet, it creates an alien process descriptor to represent the remote sending process using a standard kernel process descriptor[3] and saves the message in the message buffer field of the alien process descriptor. When the receiving process replies to the message, the reply is transmitted back to the sender as well as being saved for a period of time in the alien descriptor. If the sender does not receive a reply within the timeout period T, the original message is retransmitted by the sender's kernel. The receiving kernel filters out retransmissions of received messages by comparing the message sequence number and source process with those represented by the aliens. [illegible]

[illegible]

... network datagrams. *MoveTo* transmits the data to be moved in a sequence of maximally-sized packets to the destination workstation and awaits a single acknowledgement packet when all the data has been received. Given the observed low error rates of local networks, full retransmission on error introduces only a slight performance degradation. We have, however, implemented retransmission from the last correctly received data packet in order to avoid the pitfall of repeated identical failures arising when back-to-back packets are consistently being dropped by the receiver. The implementation of *MoveFrom* is similar except a *MoveFrom* request is sent out and acknowledged by the requested data packets, essentially the reverse of *MoveTo*.

As in the local case, major economies arise with *MoveTo* and *MoveFrom* because, by their definition, there is no need for queuing or buffering of the data in the kernel. The V kernel moves the data directly from the source address space into the network interface, and directly from the network interface to the destination address space[2].

3.4. Remote Segment Access

The message and data transfer primitives provide efficient communication of small amounts and large amounts of data, less than 32 bytes or several units of network packets. However, page-level file access requires transfer of intermediate amounts of data that is not efficiently handled by the Thoth primitives when implemented over a local network.

V file access is implemented using an I/O protocol developed for Verex [4]. To read a page or block of a file, a client sends a message to the file server process specifying the file, block number, byte count and the address of the buffer into which the data is to be returned. The server reads the data off disk, transfers it into the remote buffer using *MoveTo*, and then replies, indicating the amount of data read. In the common case of the data fitting into a single network packet, this requires [illegible]

[illegible]

at least as large as a file block, a file read operation is performed as a single low-packet exchange. In this fashion, read access to small segments (such as single block disk write operations) is handled efficiently. The *RequestWithSegment* operation illustrates the same principle in the case of a file read by combining the read request packet with the data to be returned.

Advantages of this approach include:

1. The advantages of the Thoth IPC model with the network performance of a VMTP-style file access protocol.
2. Compatibility with efficient local operation. For instance, segments may be accessed directly if in the same address space or if the recipient process operates a DMA device.
3. Use of *RecieveWithSegment* and *ReplyWithSegment* is optional and unnecessary to provide simply file access.

An apparent objection to our solution is the apparent complexity and redundancy of the message primitives. We feel the complexity may be appropriate given the previous discussion of communication and optimization. Ignoring efficiency and performance, it is not unreasonable that there be three modes of transfer: efficiency and elegance. There are applications commonly using short, simple, datagram style messages that provide a procedural interface to the message primitives. It is not inappropriate to make some compromise in the message level for efficiency.

We now turn to the discussion of the performance evaluation of the kernel. We first define the basic network penalty as a reasonable lower bound on the cost of network communication. Subsequently we discuss the efficiency of the kernel, both in terms of message passing and file access.

4. Network Penalty

Our measurements of the V kernel on processor-network configurations

1. The cost of remote operations versus the cost of the
2. The cost of the extra work V kernel remote operations

[illegible]

[illegible] operation. The network penalty is a function of the processor, the network, the network interface and the number of bytes transferred. It is the minimal time penalty for transporting the network between two software modules that could otherwise transmit the data by passing pointers. The network penalty is obtained by measuring the time to transmit a packet from the main memory of one workstation to the main memory of another and then back again and dividing the total time for the experiment by 2. The experiment is repeated a large number of times for statistical accuracy. The transfers are implemented at the data link layer with the interrupt level as low as possible to avoid including overhead imposed by the system. The assumptions of zero-time acknowledgements, good error rates and low network collision are good approximations of most local network environments.

Network penalty provides a more realistic minimum achievable time for data transfer than that suggested by the physical network speed because it includes the processor and network interface steps. For instance, a 10 Mb Ethernet can move 1000 bits from one workstation to another in 100 microseconds. However, the time to transmit the packet in the interface at the sending end and the time to transfer the packet out of the interface at the receiving end are appreciable in the time for transmission. Thus, the time for this transfer from the point of view of the communicating software modules is at least two or three times as long as that implied by the physical transfer rate.

Measurement of network penalty were made using the configurations in the following table. In all measurements, the network was essentially idle due to the unavailability of the workstations used. Table 4-1 lists our measurements of the 3 Mb Ethernet penalty for 3000 transmissions using the 8 and 32 Mb processors with zero given to milliseconds. The network penalty given in this table for the data to be transferred based on the physical bit rate of the medium, namely 3.01 milliseconds/kilobyte.

Network Penalty

Kbyte	Network Time	Network Penalty	
		8 Mb	32 Mb
0	254	630	540
512	768	1400	1310
1024	1280	2180	2100
1536	1792	2960	2870
2048	2304	3730	3650

[illegible]

[redacted]

Kernel Performance

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
	Local	Remote	Difference		Client	Server
GetTime	0.07	-	-	-	0.07	-
Send-Receive-Reply	1.00	3.18	2.18	1.60	1.79	2.30
MoveFrom: 1024 bytes	1.26	9.03	7.77	8.15	3.76	5.69
MoveTo: 1024 bytes	1.26	9.05	7.79	8.15	3.59	5.87

Table 5-1: Kernel Performance: 3 Mb Ethernet and 8 MHz Processor (times in milliseconds)

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
	Local	Remote	Difference		Client	Server
GetTime	0.06	-	-	-	0.06	-
Send-Receive-Reply	0.77	2.54	1.77	1.30	1.44	1.79
MoveFrom: 1024 bytes	0.95	8.00	7.05	6.77	3.32	4.96
MoveTo: 1024 bytes	0.95	8.00	7.05	6.77	3.17	4.95

Table 5-2: Kernel Performance: 3 Mb Ethernet and 10 MHz Processor (times in milliseconds)

view interprocess communication as transparent across machines when the speed ratio is as large. However, an alternative interpretation is to recognize that the remote operation adds a delay of less than 3 milliseconds, and that in many cases this delay is insignificant relative to the time necessary to process a request in the server. Furthermore, the reading of client workstation processor is busy with the remote kernel for only 1.44 milliseconds out of the total 2.54 milliseconds (not using the 10 MHz processor). Thus, one can offload the processor on one machine by, for instance, moving a server process to another machine if its request processing primarily requires more than 0.07 milliseconds of processor time, i.e. the additional burden on the server machine drops then and the load primarily that the 68 million

5.4. Multi-Process Traffic

[redacted]

network in this fashion. Unfortunately, our measurements of this scenario turned up a hardware bug in our 3 Mb Ethernet interface, a bug which caused many collisions to go undetected and show up as corrupted packets. Thus, the response time for the 8 MHz processor workstation in this case is 3.4 milliseconds. The increase in time from 3.18 milliseconds is accounted for almost entirely from the detection and retransmission delay (roughly one per 2000 packets) from the hardware bug. With corrected network interfaces, we believe that the network can support any reasonable level of message communication without significant performance degradation.

A more critical resource is processor time. This is especially true for workstations with no secondary storage, such as diskless workstations, where all storage needs are provided by file servers. For instance, just based on the measured remote file access overhead, a client can read 64 kilobytes from a file server in 64 one-kilobyte blocks per second, independent of the number of clients. This number is reasonably large for the entire workstation, particularly since a remote client for the server processor is utilized. File page caching can be employed to reduce [redacted]

[redacted]

serving the client process we are measuring and otherwise idle. A later section considers multi-client load on the file server.

We first describe the performance of random page-level file access.

6.1. Page-level File Access

Table 6-1 list the times for reading or writing a 512 byte block between two processes both local and remote using the 10 MHz processor. The times do not include time to fetch the data from disk but do indicate expected performance when data is buffered in memory. A page read involves the sequence of kernel operations: *Send–Receive–ReplyWithSegment*. A page write is *Send–ReceiveWithSegment–Reply*.

Random Page-Level Access

	Elapsed Time			Network Penalty	Processor Time	
Operation	Local	Remote	Difference		Client	Server
page read	1.31	5.56	4.25	3.89	2.50	3.28
page write	1.31	5.60	4.29	3.89	2.58	3.32

Table 6-1: Page-Level File Access: 512 byte pages (times in milliseconds)

The columns are to be interpreted according to the explanation given for similarly labeled columns of Tables 5-1 and 5-2. Note that the time to read or write a page using these primitives is approximately 1.5 milliseconds more than the minimal penalty for these operations.

There are several considerations that accompany the use of remote operations being slower than local operations. There are special cases of these identified for purely sequential processing. First, the extra 4.3 milliseconds that for remote operations is relatively small compared to the time cost of the file access operation itself. In particular disk access times are on the order of 20 milliseconds (assuming minimal seeking) and the system processor time of 2.5 milliseconds. This gives a total file read time of 25.5 milliseconds and a remote time of 30.1 milliseconds, making the cost of the remote operation 18 percent more than the local operation.

[redacted]

6.2. Sequential File Access

Sequential file access is the predominant pattern for file activity in most systems. Efficient file systems exploit this behavior to reduce the effect of disk latency by prefetching file pages (read-ahead) and opportunistically doing multiple pages (write-behind). File server and file transfer protocols typically implement methods to reduce the effect of network latency on sequential file access.

Using the V kernel message communication between a workstation and a file server, the file server can implement read-ahead and write-behind to reduce the effect of disk latency. However, there is no standard IPC to deal with network latency.

[redacted]

many current timesharing systems do, such program loading could achieve the same performance given in the table, independent of disk speed. Thus, we argue that *MoveTo* and *MoveFrom* with large transfer units provide an efficient program loading mechanism that is as fast as can be achieved with the given hardware.

7. File Server Issues

File server performance is a critical issue for diskless workstations. Unfortunately, we do not yet have experience with a V kernel-based file server. Thus, this section describes what we believe are the key issues and estimates performance without providing conclusive data. In general, we view the processor as the key resource to consider in file server performance because, as argued earlier, the network bandwidth is plentiful and disk scheduling and buffering issues are identical to those encountered in conventional multi-user systems.

The number of workstations a file server can support can be estimated from processor requirements. If we estimate page read or write processing overhead as roughly 3.5 milliseconds for file system processing (from LOCUS) plus 3.3 milliseconds for kernel operation (from Table 6-1), a page request costs about 7 milliseconds of processor time. Program loading appears to cost about 300 milliseconds for an average 64 kilobyte program. Estimating that 90 percent of the file requests are page requests, the average request costs 36 milliseconds. Thus, a file server based on the SUN workstation processor could support about 28 file requests a second. From this we estimate that one file server can serve about 10 workstations satisfactorily, but 30 or more active workstations would lead to excessive delays. However, a diskless workstation system can easily be expanded to handle more workstations by adding more file server machines since the network would not appear to be a bottleneck for less than 100 workstations.

For some programs, it is advantageous [illegible]

On this basis, a file server should have a [illegible]

program, i.e. it is transparent except for performance.

8. Measurements with the 10 Mb Ethernet

Our limited access to a 10 Mb Ethernet has precluded basing our measurements on this standard local network. However, some preliminary figures using the 10 Mb Ethernet indicate the effect of using a faster network and slightly faster network interfaces. First, the remote message exchange time is 2.71 milliseconds using an 8 MHz processor, roughly the time for the 10 MHz processor on the 3 Mb network and .5 milliseconds better than the 8 MHz processor on the 3 Mb network. Second, the page read time is 5.72 milliseconds. Finally, the program loading time is much improved, achieving 255 milliseconds for a 64 kilobyte load using 16 Kb transfer units. We have not identified to what degree the improvement is due to the faster network speed versus the differences in the network interface.

9. Related Work

There are a number of previous and concurrent efforts in providing communication mechanisms for distributed systems. For brevity, we compare our work with only a representative sample that characterizes the search for, and evaluation of, suitable models and implementations.

Spector's remote reference study [13] considered the feasibility of implementing remote load and store operations over a local network. Nelson's work on remote procedure calls [10] investigates network communication for procedure-based systems analogous to what the V kernel provides for message-based systems. Rashid and Robertson's Accent kernel [12] implements a message system with a number of features such as non-blocking message sending that are not provided by the V kernel. Finally, LOCUS [11] integrates network communication into a UNIX-like system in the form of transparent remote file access.

Our work has followed the pattern of Spector's and Nelson's work in using a request-response form of communication (in place of [illegible]

Our approach differs from Nelson's primarily in our use of a [illegible]

versus message-based systems, although it is not clear these differences result in any significant difference in overall performance.

The V kernel performance is roughly comparable to that of the software implementations developed by Spector and Nelson, allowing for the non-trivial differences in operation semantics and host processors. We would hypothesize that V kernel performance could be improved by a factor of 30 using microcode, similar to the improvement observed by Spector and Nelson for their primitives. Unfortunately, neither Spector nor Nelson provides results that afford a comparison with our file access results. In general, their work has concentrated on the speed of the basic mechanism and has not been extended to measure performance in a particular application setting.

In comparison to Accent, the V kernel provides a primitive form of message communication, and benefits accordingly in terms of speed, small code size and ability to run well on an inexpensive machine[5] without disk or microcode support. For instance, Accent messages require an underlying transport protocol for reliable delivery because there is no client-level reply message associated with every *Send* as in the V kernel. We do not at this time have performance figures for Accent.

LOCUS does not attempt to provide applications with general network interprocess communication but exploits carefully honed problem-oriented protocols for efficient remote file access. It is difficult to compare the two systems from measurements available given the differences in network speeds, processor speeds and measurement techniques. However, from the specific comparisons with LOCUS presented earlier, we would expect overall file access performance for the V kernel to be comparable to LOCUS running on the same machines and network.

However, the memory requirements for the V kernel are about half that of LOCUS compiled for the PDP-11 and probably more like one fourth when LOCUS is compiled for a 32-bit processor like the 68000. Thus, for graphics workstations or process control applications, for instance, the V kernel would be more attractive because of its smaller size, real-time orientation and its provision of general interprocess communication. However, the V kernel does not provide all the functionality of the LOCUS kernel which includes that of the UNIX kernel and more. When required with V, these additional facilities must be provided by server processes executing either on client workstations or network server machines.

10. Conclusions

We conclude that it is feasible to build a distributed system using diskless workstations connected by a high-speed local network to one or more file servers using the V kernel IPC. In particular, the performance study shows that V kernel IPC provides satisfactory performance despite its generality. Because the performance is so close to the lower bound given by the network penalty, there is relatively little room for improvement on the V IPC for the given hardware regardless of protocol and implementation used.

The efficiency of file access using the V IPC suggests that it can not only replace page-level file access protocols but also file transfer and remote terminal protocols, thereby reducing the number of protocols needed. We claim that V kernel IPC is adequate as a transport level for all our local network communication providing each machine runs the V kernel or at least handles the interkernel protocol. We do, however, see a place for these specific protocols in internetworking situations.

In addition to quantifying the elapsed time for various operations, our study points out the importance of considering processor requirements in the design of distributed systems. More experience and measurement of file server load and workstation file access behavior is required to decide whether file server processing is a significant problem in using diskless workstations.

The V kernel has been in use with the diskless SUN workstations, providing local and remote interprocess communication, since September 1982. It is currently 38 kilobytes including code, data and stack. The major use of the network interprocess communication is for accessing remote files. Our file servers are currently 6 VAX/UNIX systems running a kernel simulator and file server program which provides access to UNIX system services over the Ethernet using interkernel packets. A simple command interpreter program allows programs to be loaded and run on the workstations using these UNIX servers. Our experience with this software to date supports the conclusions of the performance study that we can indeed build our next generation of computing facilities [8] using diskless workstations and the V kernel.

Acknowledgements

We are indebted to all the members of the V research group at Stanford, which at this point includes two faculty members and roughly ten graduate students. In particular, we wish to thank Keith Lantz for his patient comments on a seemingly endless sequence of drafts and Tim Mann for his many contributions to the design and the implementation of the kernel. We would also like to thank the referees whose comments and suggestions helped to enhance the clarity of this paper.

References

1. F. Baskett, J.H. Howard and J.T. Montague. Task Communication in DEMOS. Proceedings of the 6th Symposium on Operating System Principles, ACM, November, 1977, pp. 23-31. Published as *Operating Systems Review* 11(5).

2. A. Bechtolsheim, F. Baskett, V. Pratt. The SUN Workstation Architecture. Tech. Rept. 208, Computer Systems Laboratory, Stanford University, March, 1982.

[5] In addition to the V kernel, the entire SUN workstation software system, including window manager, graphics support, network file server and network virtual terminal protocol, fits in 128K bytes.

3. D.R. Cheriton, M.A. Malcolm, L.S. Melen and G.R. Sager. "Thoth, a Portable Real-time Operating System." *Comm. ACM 22*, 2 (February 1979), 105-115.

4. D.R. Cheriton. Distributed I/O using an Object-based Protocol. Tech. Rept. 81-1, Computer Science, University of British Columbia, 1981.

5. D.R. Cheriton. *The Thoth System: Multi-process Structuring and Portability*. American Elsevier, 1982.

6. D.R. Cheriton, T.P. Mann and W. Zwaenepoel. V-System: Kernel Manual. Computer Systems Laboratory, Stanford University

7. Digital Equipment Corporation, Intel Corporation and Xerox Corporation. The Ethernet: A Local Area Network - Data Link Layer and Physical Layer Specifications, Version 1.0.

8. K.A. Lantz, D.R. Cheriton and W.I. Nowicki. Third Generation Graphics for Distributed Systems. Tech. Rept. STAN-CS-82-958, Department of Computer Science, Stanford University, February, 1983. To appear in *ACM Transactions on Graphics*

9. R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Comm. ACM 19*, 7 (July 1976), 395-404. Also CSL-75-7, Xerox Palo Alto Research Center, reprinted in CSL-80-2.

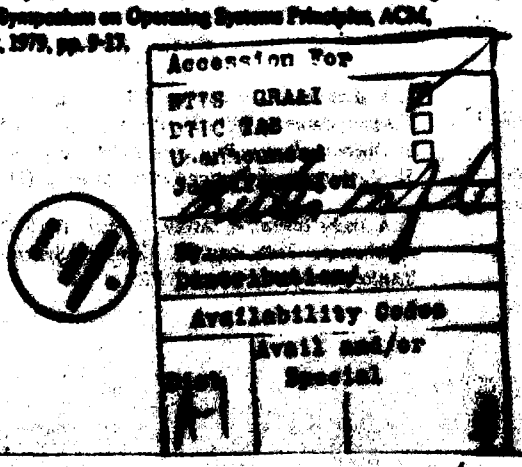
10. B.J. Nelson. *Remote Procedure Call*. Ph.D. Th., Carnegie-Mellon U., 1981. published as CMU technical report CMU-CS-81-119

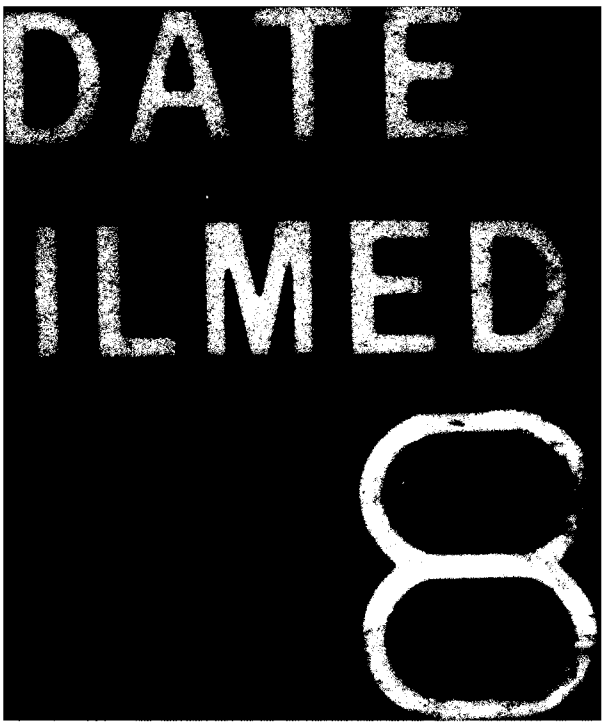
11. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G.Rudisin, G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. Proceedings of the 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 169-177.

12. R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. Proceedings of the 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 64-75.

13. A. Spector. "Performing Remote Operations Efficiently on a Local Computer Network." *Comm. ACM 25*, 4 (April 1982), 246-260.

14. D. Swinehart, G. McDaniel and D. Boggs. WFS: A Simple Shared File System for a Distributed Environment. Proceedings of the 7th Symposium on Operating Systems Principles, ACM, December, 1979, pp. 9-17.




DATE
ILMED
8